Quantstamp Security Assessment Certificate

February 26th 2021 — Quantstamp Verified

LinumLabs Swarm Curve

This smart contract audit was prepared by Quantstamp, the protocol for securing smart contracts.

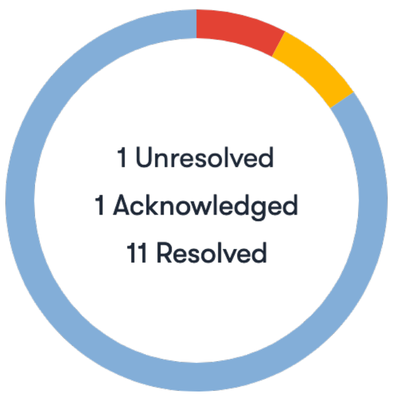
Executive Summary

Type	ERC20 Token
Auditors	Kacper Bąk, Senior Research Engineer Ed Zulkoski, Senior Security Engineer Jose Ignacio Orlicki, Senior Engineer
Timeline	2021-02-05 through 2021-02-26
EVM	Byzantium
Languages	Solidity, Javascript
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	README
Documentation Quality	High
Test Quality	Medium

Source Code

Repository	Commit
swarm-curve (original repo name)	9a9a0ae (initial audit)
bzzaar (new name on reaudit)	dc28d88 (final audit)

Total Issues	**13** (11 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	**1** (1 Resolved)
Low Risk Issues	**1** (1 Resolved)
Informational Risk Issues	**5** (3 Resolved)
Undetermined Risk Issues	**6** (6 Resolved)



High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.

Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Resolved	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

After audit: Quantstamp has identified several issues spanning overall severity levels, in the `swarm-curve` codebase. Overall the documentation quality is very good, excepting the helper to compute token prices, and the code coverage of the test is very high with the exception of one module (`Eth_broker.sol`). The test suite was very comprehensive with 46 tests. However, we were able to identify a modest number of 50 assertions in the test files, which indicates that not all of the functionality is accurately tested. It is of utmost importance for any production-ready project to have a code coverage as close as possible to 100% and a high number of assertions in order to ensure that all the functionality of the smart contracts has been tested. Finally, a few deviations from best practices and code documentation issues were found during the audit. We strongly recommend that all of these issues be addressed before deploying the code on the Ethereum mainnet.
Update: We have surveyed fixes from the initial review report based on the changes here. All issues appear to be resolved, with the exception of a token swap return amount, not checked. No new issues were found to be presented within the fixed commit sent. Documentation and inline comments have been reexamined and improved in almost all areas, including critical number arithmetic for pricing. Old copied code was dispensed with, and redundant functions and required checks have been abstracted and concentrated in on `_commonMint()`.

ID	Description	Severity	Status
QSP-1	Possible Transfer to 0x0 / Contract Address	^ Medium	Fixed
QSP-2	Reentrancy	v Low	Fixed
QSP-3	Allowance Double-Spend Exploit	○ Informational	Mitigated
QSP-4	Unexpected Ether	○ Informational	Acknowledged
QSP-5	Unchecked constructor arguments	○ Informational	Fixed
QSP-6	Incorrect/Missing Visibility	○ Informational	Fixed
QSP-7	Unchecked Return Value	○ Informational	Unresolved
QSP-8	Broken internal state	? Undetermined	Fixed
QSP-9	Unsound arithmetic on the Bonding curve	? Undetermined	Fixed
QSP-10	Unsound arithmetic related to `_minDaiSellValue`	? Undetermined	Fixed
QSP-11	Unused state variables or constants	? Undetermined	Fixed
QSP-12	Irregular or out-of-date interface for `burn`	? Undetermined	Fixed
QSP-13	Unused local variable	? Undetermined	Fixed

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.7.0
- Mythril v0.22.16

Steps taken to run the tools:

1. Installed the Slither tool: `pip install slither-analyzer`
2. Run Slither from the project directory: `slither .`
3. Installed the Mythril tool from Pypi: `pip3 install mythril`
4. Ran the Mythril tool on each contract: `myth -x path/to/contract`

Findings

QSP-1 Possible Transfer to 0x0 / Contract Address

Severity: Medium Risk

Status: Fixed

File(s) affected: `Eth_portal_flat.sol`

Description: It is rarely desirable for tokens to be sent to the `0x0` address (intentional token burning is a notable exception) nor to the contract itself. However, these mistakes are often made due to human errors. Hence, it's often a good idea to prevent these mistakes from happening within the smart contract itself.
Internal state variable `daiAddress_` is returned in `getPath()` as a usable token address but is never initialized. Duplicated with `dai_`.

Recommendation: Initialize this address in `constructor()` or use only `dai_` if variables are duplicated.

QSP-2 Reentrancy

Severity: Low Risk

Status: Fixed

File(s) affected: `Eth_portal_flat.sol`, `Curve.sol`

Description: A reentrancy vulnerability is a scenario where an attacker can repeatedly call a function from itself, unexpectedly leading to potentially disastrous results. Here's a basic example representing the very attack which impacted The DAO in 2016:

```
    ```
    function withdraw_with_reentrancy(uint256 _amount) public {
        msg.sender.call.value(_amount)();   // ATTACKER CAN CALL AGAIN BEFORE
                                            // FUNCTION TERMINATES because `call`
                    // allows msg.sender to execute any code using fallback function
        balances[msg.sender] -= _amount;   // Balance only updated AFTER funds withdrawn
    }
    ```
```

The content of `Eth_portal_flat.sol` mostly resembles `Eth_broker.sol`, however, a mutex lock is not used around the `mint` and `burn` functions. This may introduce unforeseen reentrancy attack vectors.

Recommendation: Add the mutex lock modifier to each function. There is a very popular implementation in OpenZeppelin Contracts `openzeppelin-contracts/contracts/utils/ReentrancyGuard.sol`.

QSP-3 Allowance Double-Spend Exploit

Severity: Informational

Status: Mitigated

File(s) affected: `Token.sol`

Description: As it presently is constructed, the contract is vulnerable to the allowance double-spend exploit, as with other ERC20 tokens.

Exploit Scenario:

1. Alice allows Bob to transfer `N` amount of Alice's tokens (`N>0`) by calling the `approve()` method on `Token` smart contract (passing Bob's address and `N` as method arguments)
2. After some time, Alice decides to change from `N` to `M` (`M>0`) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve()` method again, this time passing Bob's address and `M` as method arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls the `transferFrom()` method to transfer `N` Alice's tokens somewhere
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer `N` Alice's tokens and will gain an ability to transfer another `M` tokens
5. Before Alice notices any irregularities, Bob calls `transferFrom()` method again, this time to transfer `M` Alice's tokens.

Recommendation: The exploit (as described above) is mitigated through use of functions that increase/decrease the allowance relative to its current value, such as `increaseAllowance()` and `decreaseAllowance()`.
Pending community agreement on an ERC standard that would protect against this exploit, we recommend that developers of applications dependent on `approve()` / `transferFrom()` should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value. Teams who decide to wait for such a standard should make these recommendations to app developers who work with their token contract.

QSP-4 Unexpected Ether

Severity: Informational

Status: Acknowledged

File(s) affected: `Eth_broker.sol`

Description: Smart contracts, though they may not expect it, can receive ether forcibly. This may affect the operation of the smart contract in unpredictable ways. For `Eth_broker`, at L237, if ETH is sent to this contract (either before creation or forcibly), `msg.sender` may receive more ETH than expected.

QSP-5 Unchecked constructor arguments

Severity: *Informational*

Status: Fixed

File(s) affected: `Eth_broker.sol`

Description: Should ensure that `_bzzCurve`, `_daiToken` and `_router02` are non-zero.

QSP-6 Incorrect/Missing Visibility

Severity: *Informational*

Status: Fixed

File(s) affected: `Curve.sol`

Description: The visibility of a function or field changes determines how it can be accessed by others. Using the right visibility ensures optimal gas costs and reduces the possibility of attacks. The four types of visibility are: * `external` - can be called by any other contract (but not within the contract itself) * Useful for optimizing gas costs * `public` - can be called anywhere * `internal` - can only be called within the contract, and inherited contracts will inherit this functionality * Useful for creating functions that should not be called by anyone else * `private` - can only be called within the contract, cannot be inherited
Use `external` declaration for functions not used in other functions. Functions only called externally by other contracts or users can be only declared as `external`, gas is saved and attackers are given less internal functions and control in case of vulnerabilities. This way they cannot be called from other `internal` or `public` functions.
Functions affected: `isCurveActive()`, `requiredCollateral()`, `init()`, `mint()`, `mintTo()`, `redeem()`, `shutDown()`.

QSP-7 Unchecked Return Value

Severity: *Informational*

Status: Unresolved

File(s) affected: `Eth_broker.sol`

Description: Most functions will return a `true` or `false` value upon success. Some functions, like `send()`, are more crucial to check than others. It's important to ensure that every necessary function is checked.
Functions affected: `Eth_broker` L224, L233, L235, L285, L294, L296, L354, L369, L374.

Recommendation: Always check return values of functions and handle them accordingly.
Update: Although is partially solved, we recommend adding an extra `require()` after the call to `router_.swapETHForExactTokens()` on L420 to check that the DAI received is at least `daiNeeded`.

QSP-8 Broken internal state

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Curve.sol`

Description: Asset supply changes usually need to adjust tokenomics internal state. Make sure the constants `bzzscale_` and `openMarketSupply_` are in line with the token supply. Make sure `openMarketSupply_` is adjusted whenever `mint()` or `redeem()` is executed.

Exploit Scenario: If the attacker redeems a big number of tokens then the balance can be smaller than `openMarketSupply_`.

Recommendation: Track tokenomics in `mint()` and `redeem()`.

QSP-9 Unsound arithmetic on the Bonding curve

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Curve.sol`

Description: There is not enough documentation or context to understand the rationale for the equation computed by `_helper()`. This is directly related to `_primitiveFunction` and the bonding curve of choice.

Recommendation: Elaborate in detail the bonding curve arithmetic and how is calculated in the implementation.

QSP-10 Unsound arithmetic related to `_minDaiSellValue`

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Eth_broker.sol`

Description: In `redeem`, on L366-367 we have the following:

```
// Getting expected ETH for DAI
uint256 ethMin = sellRewardDai(_minDaiSellValue);
```

From documentation or context it cannot be inferred the rationale for using `_minDaiSellValue` instead of `dai_.balanceOf(address(this))`, the latter possibly being higher than `_minDaiSellValue`.

QSP-11 Unused state variables or constants

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Curve.sol`

Description: The state variables `reserveRatioDenominator_` and `scale_`, which relate to the bonding curve price computations, are unused. It is unclear if the related functions are implemented correctly. It is also unclear if these state variables should be constant or not.

QSP-12 Irregular or out-of-date interface for `burn`

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Eth_portal_flat.sol`

Description: `burn` may be using an out-of-date interface. On L326, `curve_.burn` is used. While this matches the interface `I_Curve` at the top of the file, the `burn` function does not exist in `Curve.sol` nor `I_Curve.sol`. It is not clear if this flattened file is up-to-date.

Recommendation: Ensure that flattened files are up-to-date.

QSP-13 Unused local variable

Severity: *Undetermined*

Status: Fixed

File(s) affected: `Curve_flat.sol`

Description: On function `_initializeCurve()` there are two priced local variables: `price` and `initial_price`. Only `price` is used. From documentation or context cannot be determined what is the use of `initial_price` that comes from `_spotPrice()`.

Code Documentation

1. In `Eth_broker.redeem`, the comment on L339: "Checking that this amount is not more than the maximum spends amount" appears to be a copy+paste error from the `mint` functions. It should instead say "Checking that this amount is at least the min sell amount".
2. In `Curve._initializeCurve`, the comment "@return initial_price The starting price of the token" does not match the code; the function only returns the averaged `price`. **Update: fixed**

Adherence to Best Practices

We have identified the following deviations from best-practices:

1. Code duplication in `Eth_broker.mint()` and `Eth_broker.mintTo()`. **Update: fixed**
2. Code duplication in `Curve.mint()` and `Curve.mintTo()`. **Update: fixed**
3. There are duplicate checks for `Eth_broker.redeem` and `Curve.redeem`. For example, both compute the `sellReward` and then check `reward >= _minCollateralReward`. This costs extra gas. It is unclear why the check in `Eth_broker.redeem` is needed. **Update: fixed**
4. Some naming conventions are inconsistent and error-prone, such as internal state variables `collateralToken_`and state variables `_status`. **Update: fixed**

Test Results

Test Suite Results

49 passing (2m)

```
$ yarn test
yarn run v1.22.10
$ etherlime test --solcVersion=0.5.0 --output=none --timeout 100000 --gas-report=true
(node:11722) Warning: Accessing non-existent property 'VERSION' of module exports inside circular dependency
(Use `node --trace-warnings ...` to show where the warning was created)
(node:11722) Warning: Accessing non-existent property 'INVALID_ALT_NUMBER' of module exports inside circular dependency
(node:11722) Warning: Accessing non-existent property 'INVALID_ALT_NUMBER' of module exports inside circular dependency

  ✓ Broker tests
   Mock router tests
      ✓ Gets Amounts In returns correct values (18ms)
      ✓ get weth address expected (16ms)
      ✓ Swap ETH for exact tokens (DAI) test (91ms, 58325 gas)
   broker view tests
      ✓ Swap exact tokens (DAI) for ETH test (197ms, 124821 gas)
      ✓ buy price expected (33ms)
      ✓ sell reward expected (42ms)
      ✓ sell reward dai expected (20ms)
      ✓ Get path expected (28ms)
   broker tests
      ✓ get time works as expected (13ms)
      ✓ mint slippage check (44ms)
      ✓ mint balance checks (1077ms, 174613 gas)
      ✓ mintTo balance checks (1206ms, 175643 gas)
      ✓ burn fails without approval (44ms)
  ✓ Curve tests
   Curve pre-mint collateral tests
      ✓ burn balance checks (627ms, 269061 gas)
      ✓ Pre-mint can sell down curve (partial) (464ms, 127479 gas)
      ✓ Pre-mint can sell down curve (almost whole pre-mint) (456ms, 127371 gas)
   Curve slippage tests
      ✓ Price can slide back to pre-mint supply (918ms, 291772 gas)
      ✓ Price cannot exceed max spend (buy) (689ms, 259375 gas)

   Curve calculations tests
      ✓ Price cannot exceed max spend (sell) (1535ms, 500361 gas)
```

```
      ✓ Cannot buy for 0 (14ms)
      ✓ Tokens correctly minted on buy (520ms, 164293 gas)
      ✓ Tokens correctly minted on mintTo buy (571ms, 165235 gas)
      ✓ Cannot sell for 0 (22ms)
      ✓ Tokens correctly burnt on sell (1071ms, 306772 gas)
      ✓ Open market price correct (16ms)
      ✓ Bonded token address (9ms)
    Curve shut down tests
      ✓ Collateral token address (8ms)
      ✓ Can shut down the curve (66ms, 25545 gas)
      ✓ Non owner cannot shut down the curve (16ms)
    Curve ownership tests
      ✓ Once shut, blocked functions cannot be accessed (106ms, 25545 gas)
      ✓ Owner is set correctly (13ms)
    Curve bonded token tests
      ✓ Ownership can be transferred correctly (61ms, 31479 gas)
      ✓ Bonded tokens burnt outside of curve blocked (532ms, 208625 gas)
    Curve safe math tests
      ✓ Bonded tokens can only be burnt by minter (264ms, 208118 gas)
      ✓ Reverts on buy price if amount too large (14ms)
      ✓ Reverts on buy price if amount at max supply (16ms)
      ✓ Reverts on buy price if amount at max supply (16ms)
      ✓ Buy price if amount 0 is 0 (19ms)
      ✓ Reverts on buy price of 1 dec is not 0 (17ms)
      ✓ Reverts on sell reward if amount too large (13ms)
      ✓ Reverts on sell reward if amount at max supply (12ms)
      ✓ Reverts on sell reward if amount 0 (19ms)
  ✓ Curve Calculations Tests
    Curve pre-init tests
      ✓ Reverts on sell reward of 1 dec is not 0 (16ms)
      ✓ Pre-mint cost consistent (52ms)
    Curve post-init tests
      ✓ Spot price before init (15ms)
      ✓ Helper is correct (10ms)
      ✓ Price at start of curve (48ms)
  ✓ Curve pre-mint tests
    Curve initialisation tests
      ✓ Withdraw reward at start (467ms, 164315 gas)
      ✓ Can't set up curve with less than expected pre-mint (122ms, 46498 gas)
      ✓ Curve set up with pre-mint (exact) (320ms, 277337 gas)
      ✓ Curve set up with pre-mint (above expected) (351ms, 277385 gas)
      ✓ Can buy tokens (856ms, 441630 gas)
      ✓ Can sell tokens (1237ms, 584109 gas)
      ✓ Cannot double initialize (312ms, 277337 gas)
  ✓ Token Tests
    Parent input validation checks
      ✓ Cannot initialize if curve is not minter (170ms, 69139 gas)
      ✓ (detailed) Correct deployment (154ms)
    ✓  ✓ (cap) Can't deploy a 0 cap (36ms)

  Total Gas Used: 5382183


  57 passing (54s)

✓  Done in 81.01s.
```

Code Coverage

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	81.52	51.61	78.69	81.27	
Curve.sol	100	76.92	100	100	
Curve_test.sol	100	100	100	100	
Eth_broker.sol	97.78	58.33	92.31	97.87	372
I_Curve.sol	100	100	100	100	
I_Token.sol	100	100	100	100	
I_router_02.sol	100	100	100	100	
Mock_dai.sol	54.1	29.17	50	53.23	… 516,524,540
Mock_router.sol	81.82	42.86	83.33	81.82	22,26,27,28
Token.sol	78.57	47.22	80	77.91	… 414,415,649
All files	**81.52**	**51.61**	**78.69**	**81.27**	

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

```
59ceffb96e893acdae0b0fe0a51bd1a6f99345b8c314e3ad3c0c3841e97bfb10 ./contracts/Token.sol
15782638f4cc4e45b5d04e7a8534f272246c9945c5d748fea4295ee119dea4d3 ./contracts/Curve_test.sol
eb3bbb2891831f2139d7573e0a0f4c7a8c6a2089394989a75282cb15ae3517d6 ./contracts/I_Curve.sol
1823738d583b6a0873fd48a8b194fd7b4cf3e77d2ce0a18580a0fdff95fd98ef ./contracts/Mock_router.sol
f8da830939d0cb5ea79c47d6268e779050b48ad5bcc46aea0aadce670d8e7c48 ./contracts/Mock_dai.sol
8e477c74c5fd8d56ca5beef59919ee9c27d644b68df0fbf94f2f3c97eb60a750 ./contracts/Eth_broker.sol
1d623e08184cb5e49ad3416c6b6f754caf9cc5f6a3b9cf62941ab907c8260ad1 ./contracts/Curve.sol
1ef4639e61249b393f50b9523685e3f0a7698db09b8d230785869cf98f92cb76 ./contracts/I_router_02.sol
972dfabc1cbf1e21400c82413681784bbc5c1b56232ea11ce645aedadedfd95a ./contracts/I_Token.sol
```

Tests

```
2edf2f6d8fb49ddc56d1ff7b189a1329c275b52635654de3e3aa812932de33d2 ./test/token.test.js
bc05e80e1598d5373c1dcbcf62c884176076efafbcfe2e7f692d2f26962aa105 ./test/broker.test.js
2d06fb2fe2be4f5e7d930931179260c09a25d0dc1fd8ba6936f906dfba840d7e ./test/curve.test.js
cbba6ef098039d84b0e609dd16382d60637f392e4a10465d761dde0059805273 ./test/curve_pre_mint.test.js
a267e046797977fe1b09a4d9d571713b433d8bdd4762c46f9ae89bece9116a4c ./test/curve_calc.test.js
88fc5b52ef2cadf9232649f7460929f9ed7b0109ce95b4e6acad479c99991f54 ./test/settings.test.js
```

Changelog

- 2021-02-08 - Initial report
- 2021-02-25 - Updated report based on commit dc28d883

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure blockchain platforms at scale using computer-aided reasoning tools, with a mission to help boost the adoption of this exponentially growing technology.

With over 1000 Google scholar citations and numerous published papers, Quantstamp's team has decades of combined experience in formal verification, static analysis, and software verification. Quantstamp has also developed a protocol to help smart contract developers and projects worldwide to perform cost-effective smart contract security scans.

To date, Quantstamp has protected $5B in digital asset risk from hackers and assisted dozens of blockchain projects globally through its white glove security assessment services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Quantstamp's collaborations with leading academic institutions such as the National University of Singapore and MIT (Massachusetts Institute of Technology) reflect our commitment to research, development, and enabling world-class blockchain security.